MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLEGENCE LABORATORY

A.I. Memo 507

January 1979

A Hypothetical Monologue Illustrating The Knowledge Underlying Program Analysis

Howard E. Shrobe, Richard C. Waters, and Gerald J. Sussman

Abstract

Automated Program Analysis is the process of discovering decompositions of a system into sub-units such that the behavior of the whole program can be inferred from the behavior of its parts. Analysis can be employed to increase the explanatory power of a program understanding system. We identify several techniques which are useful for automated program analysis. Chief among these is the identification and classification of the macro-scale units of programming knowledge which are characteristic of the problem domain. We call these *plans*. This paper presents a summary of how plans can be used in program analysis in the form of a hypothetical monologue. We also show a small catalogue of plans which are characteristic of AI programming. Finally, we present some techniques which facilitate plan recognition.

*This paper was adapted from a proposal to the National Science Foundation.

This report describes research conducted at the Artificial Intelligence Laboratory and at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Support for the Artificial Intelligence Laboratory's Artificial Intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643. Support for the Laboratory for Computer Science's research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0661.

The programmer's apprentice project at MIT [Rich & Shrobe 1976,78] is attempting to develop a knowledge based assistant for the expert programmer. Most other research in this direction has centered on investigating the knowledge which underlies program synthesis. Notable among such systems has been PECOS [Barstow 1977] which catalogues a significant fraction of routine programming knowledge in a form suitable for use in a refinement oriented program synthesizer.

In contrast, the apprentice project has spent considerable effort studying the analysis process. Analysis is the process of segmenting a mechanism such that the behavior of the sub-parts can be clearly identified and such that the behavior of the overall mechanism can be inferred from the behavior of the parts. Analysis often requires multiple decompositions. Different properties of the overall mechanism are often revealed more clearly in one particular viewpoint. For example, in electrical circuit analysis one makes one decomposition for the DC analysis and a second decomposition for the incremental analysis.

The kind of analysis performed by the programmer's apprentice system is quite different from the kind of reasoning which is done as part of program verification. The goal of verification is to certify the correctness of a program. In contrast, the goal of the analysis discussed here is to provide common sense explanations of how a program functions and to provide a basis for understanding the consequences of a modification. In light of this goal, the reasoning in analysis need not be as formal or exact as the reasoning in verification. For example, in the analysis of physical systems inexact models are used routinely to simplify the task of understanding a system.

Although there has been considerable AI research on the automated analysis of physical systems, there has been considerably less research on program analysis. In this paper we present, in the form of a monologue, an illustration of some of the knowledge and processing which a good programmer would employ to analyze a reasonably complex set of LISP functions.

Structure of the Apprentice's Analysis System

The apprentice represents programs using abstract "plan diagrams" which are a programming langauge independent representations of procedures. All identifiers in the plan formalism are completely local. The plan diagrams consist solely of: (1) Segments, described either by their Input/Output specification or by another plan diagram; (2) Data Flow Links which indicate the flow of information; and (3) Control Flow Links constraining the order of segment execution. In this paper we show plan diagrams in a pictorial fashion, the apprentice represents them using assertions in a data base.

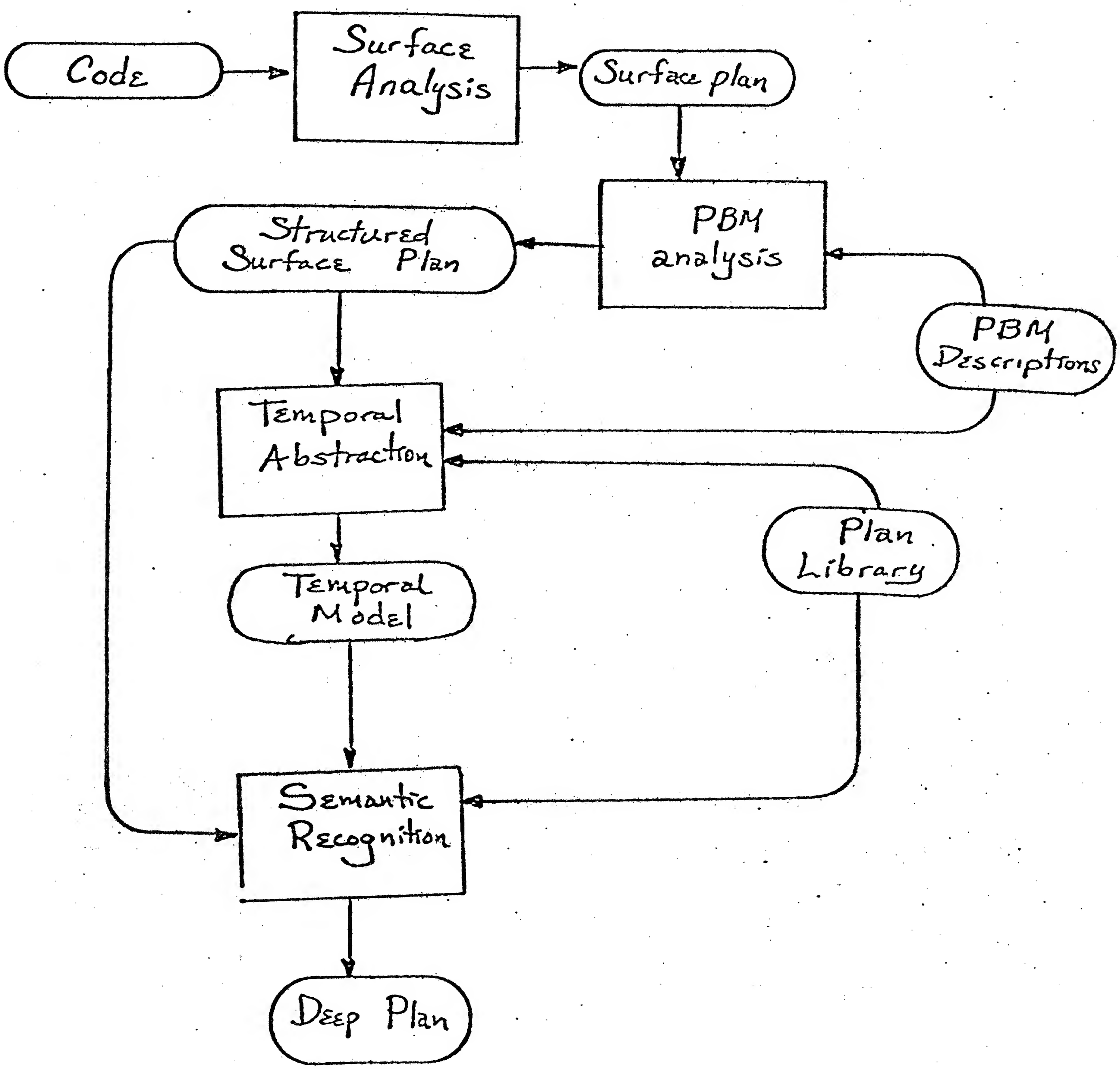

Figure 1: The flow of information in the analysis component of the apprentice system.

The flow of information within the analysis component of the apprentice system is shown in Figure 1. The processing steps are additive; the apprentice never throws away information, but rather builds an increasingly rich description of the code. Although each stage of analysis adds a new layer of annotation, there is no strict barrier between these layers. Sometimes deductions

triggered by the information existing in one layer will propagate information to and induce deductions in other layers.

The analysis process used by the programmer's apprentice takes place in four steps. The first step is surface flow analysis which reformulates the program in the plan language of data and control flow. The only segmentation which is produced in this stage is that which is obvious from the way the code is written. For example, each subroutine is made a seperate segment.

The second step of analysis is Plan Building Method analysis. Plan Building Methods (PBMs) [Waters, 1978] correspond to ways in which smaller plans are combined to form larger ones. The primary value of PBM analysis is that it imposes hierarchical structure on the plans being analyzed. Three PBMs are used to segment straight-line code. These correspond to the standard notions of a conditional expression, conjunction, and composition. Four other PBMs are used to analyze recursive code (including loops). The basic idea behind these PBMs is that recursive structures can be broken apart into stereotyped fragments of recursive behavior. Recursive program segments are broken apart into *augmentations* which create and use sequences of values, *terminations* which test sequences of values and control the termination of the segment, and *filters* which restrict sequences of values based on a test.

PBM analysis is followed by temporal analysis [Shrobe, 1978] [Waters, 1978] which constructs temporal models. Temporal analysis examines the entire potential history of a computation, segmenting the occurances of various segments into causally coherent units. One example of this is closely related to the PBMs for recursive segments. PBM analysis suggests the separation of generators and consumers within recursive programs. Each generator is then modelled as a segment which produces a *temporal collection of values* as its output. This temporal collection is then used as an input by the consumers. A temporal collection is a set of objects which does not exist as a unified data structure but which is produced by the repeated application of a segment. For example, a program which builds a list of the terminal nodes of a tree can be modelled as the composition of a *Tree-Traversal* (Car-Cdr recursion) which visits every node of the tree and a *Cons Accumulation* which receives each terminal node as it is generated and adds the node to the list of already accumulated terminal nodes.

Tree Traversal and Cons Accumulation are examples of *standard plans*, extremely common patterns of program behavior. A relatively small number of such standard plans constitute the bulk of the apprentice's knowlege of programming. These have been catalogued in a *plan library* [Rich, forthcoming] which is the apprentice's knowledge base. The causal aggregation performed during PBM and temporal analysis breaks the plan for a program into units which correspond quite closely to standard plans of the library. This facilitates the final stage of the analysis process. In

this stage (called *plan recognition*), the program is divided up in such a way that fragments of it can be mapped onto standard plans in the library.

Plan recognition is not based on a monolithic grammar which parses the program as a whole. Rather it extracts structure from fragments of the program. It is not necessary for our system to completely understand a program as a single unit in order for it to be useful. Evan a partial analysis provides a framework for organizing additional information supplied by the user. For example, even though the apprentice does not have any expertise in the area of algorithmic analysis, it can store and index information about time and space tradeoffs in such a way that it can be retrieved and used with adequate flexibility.

The annotation developed during program analysis is coordinated by means of a dependency-directed reasoning system [de Kleer et. al. 1977, Shrobe, 1978]. Every assertion in the system's data base is associated with a *justification* explaining why the assertion is believed. These are managed by a Truth Maintenance System [Doyle, 1978]. An important feature of this system is its ability to make an assumption in such a way that if the assumption is ever discovered to be invalid any conclusion which depended on the assumption will automatically be removed from the data base. This makes it easy for the system to operate in a hypothesis-driven manner.

A Detailed Scenario

The following scenerio is a hypothetical monologue which illustrates the kind of knowledge and reasoning which is required in order to perform plan recognition. The code which is analyzed in this scenario is a procedural deduction system. Such systems utilize two data bases in tandem. The first is a data base of facts, while the second is a data base of demons. A demon consists of a procedure and a pattern. The procedure is invoked any time a fact which matches the pattern is entered into the data base. Each data base is completely inverted so that pattern directed retrieval may be done based on any fragment of an item. The code in the example is written in MACLISP.

Before beginning the monologue, we present several of the plan library's standard plans. These will be used extensively during the monologue. In the diagrams, segments are represented as boxes, data flow is represented by solid arrows, and control flow is represented by cross hatched arrows. Recursion is indicated by a looping line connecting two boxes. This indicates that the two boxes have exactly the same internal plan.

1. List enumeration or Cdr recursion (Figure 2) The key features of this plan-type are the null-test, and the data flow from the cdr segment to the singly recursive call.

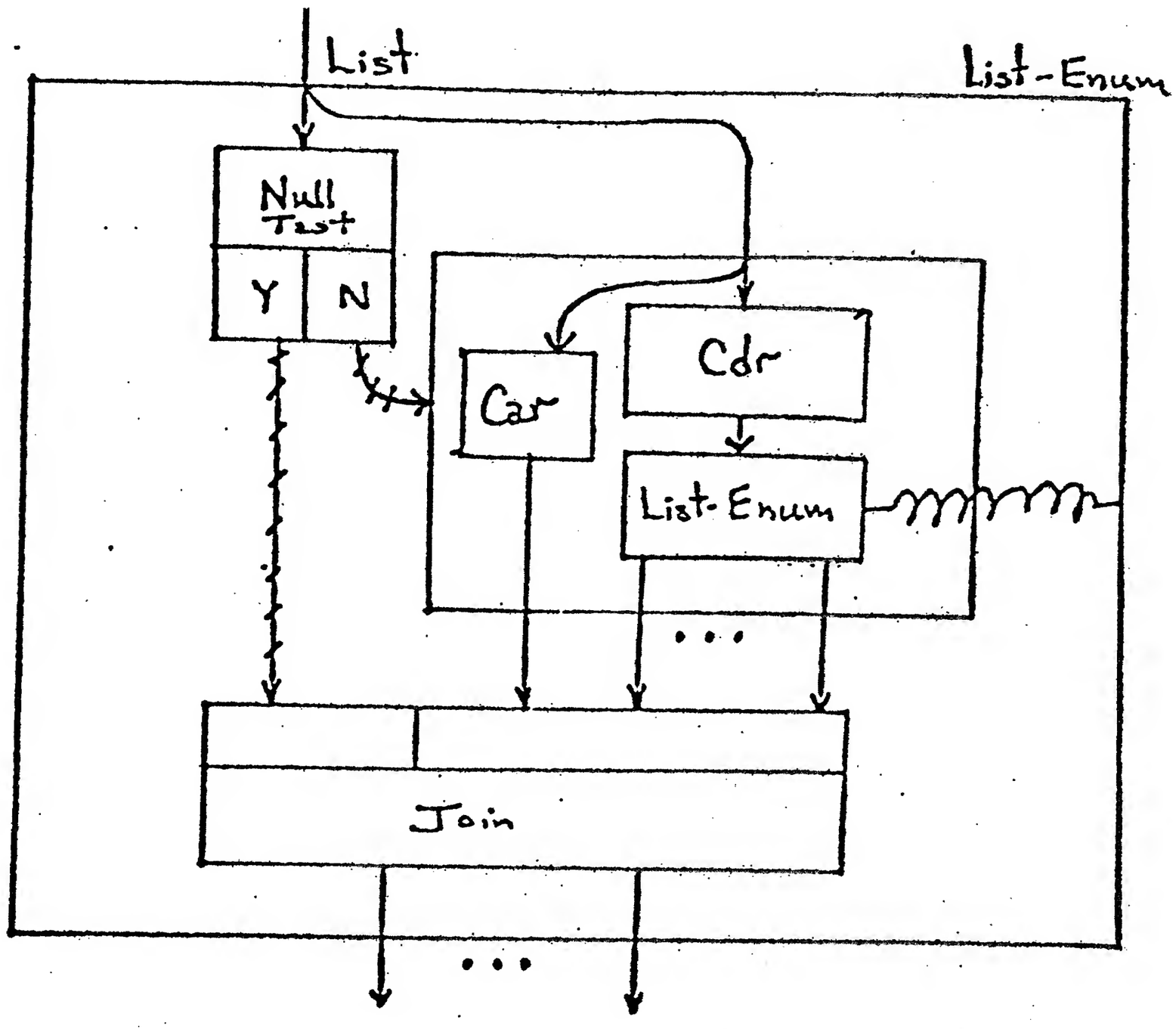

Figure 2: The plan diagram for List Enumeration.

2. Binary Tree Traversal or Car/Cdr recursion (Figure 3) This plan-type is characterized by an atom test, and data flows from each of a Car and a Cdr segment to two recursive calls.

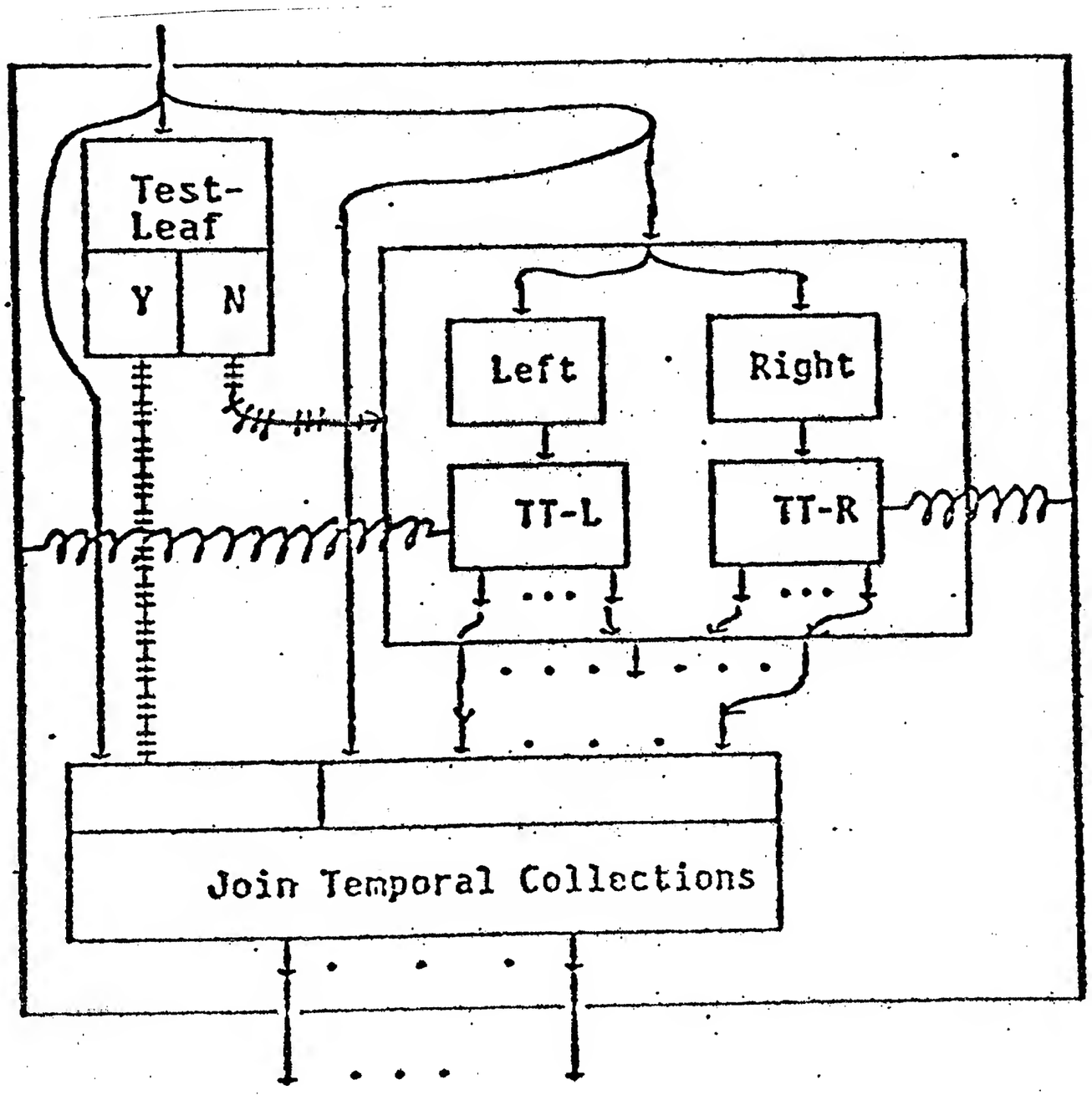

Figure 3: The plan diagram for Tree Traversal.

3. Sequential Accumulation (Figure 4) This plan-type is most easily understood in its temporal model. From the temporal viewpoint, sequential accumulation is characterized by a temporal collection input, a cascade of accumulator segments, an initialization input (typically an identity element for the operator used in the accumulation) and an output flowing from the last accumulator segment.

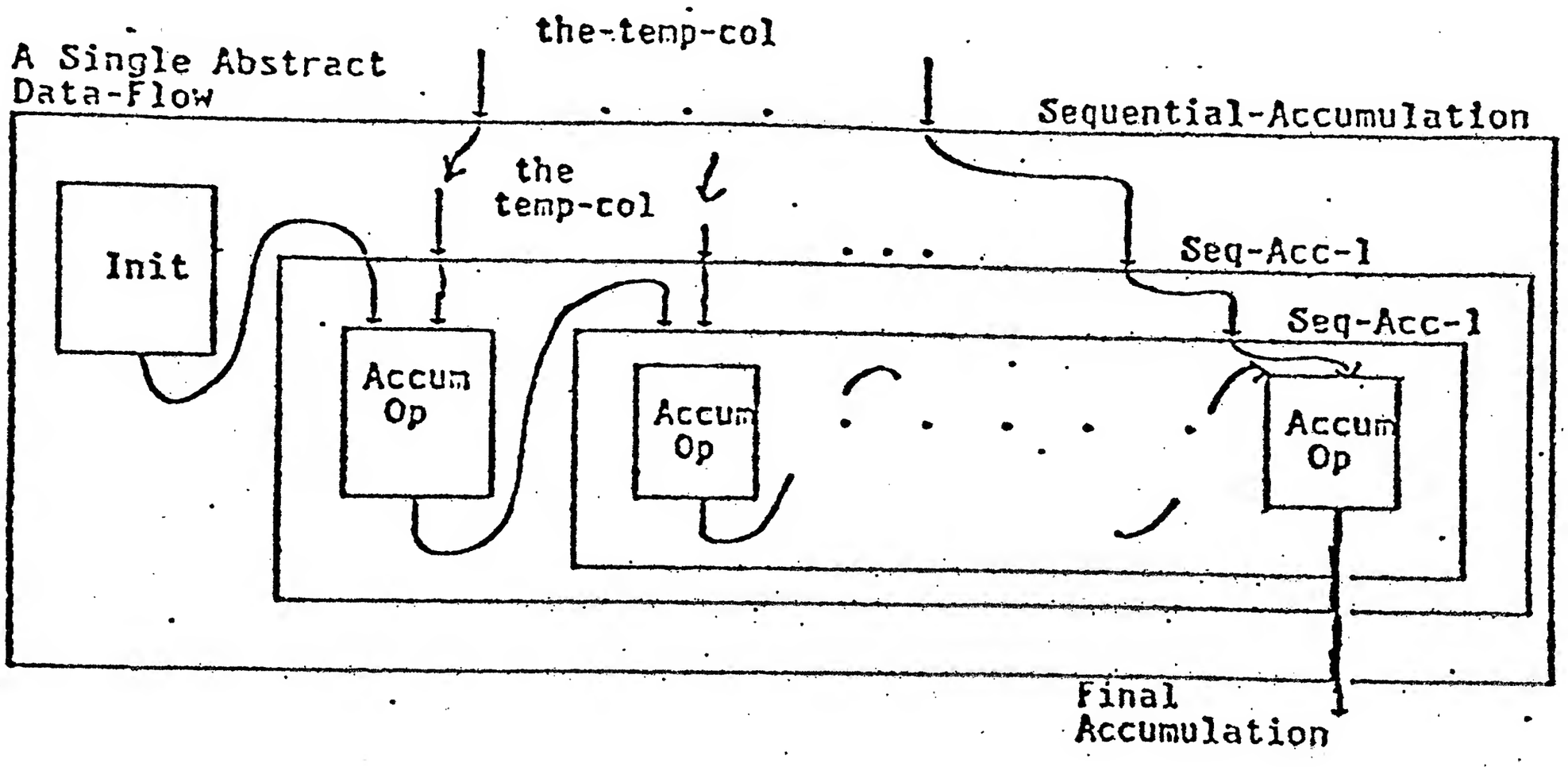

Figure 4: Sequential Accumulation with a Temporal Collection input.

3a. Sequential Cons Accumulation This is a specialization of Sequential Accumulation in which the accumulator segment is "CONS". The initialization is typically "NIL".

3b. Counting This is another specialization of Sequential Accumulation in which the accumulator is an Add-One segment and the initialization is 0.

4. Filtered Accumulation (Figure 5) This plan-type is also understod most easily in its temporal model. A Filtered Accumulation plan consists of three segments when modelled temporally. (i) A generator: a segment which inputs a normal input and produces a temporal collection output. (ii) A filter, which is a collection of test segments. The filter produces a smaller temporal collection as its output which consists of exactly those members of the input which satisfy the test. (iii) An accumulator which takes in a temporal collection and produces a single data structure as its output. The accumulator contains a cascade of accumluation operator sub-segments in one-to-one correspondence with the members of the temporal collection input. This plan has a large number of specializations depending on what generator, filter and accumulator are used. Notice that the generator and the accumulator are initilized independently; these steps are called the Enumeration Initialization and the Accumulation Initialization.

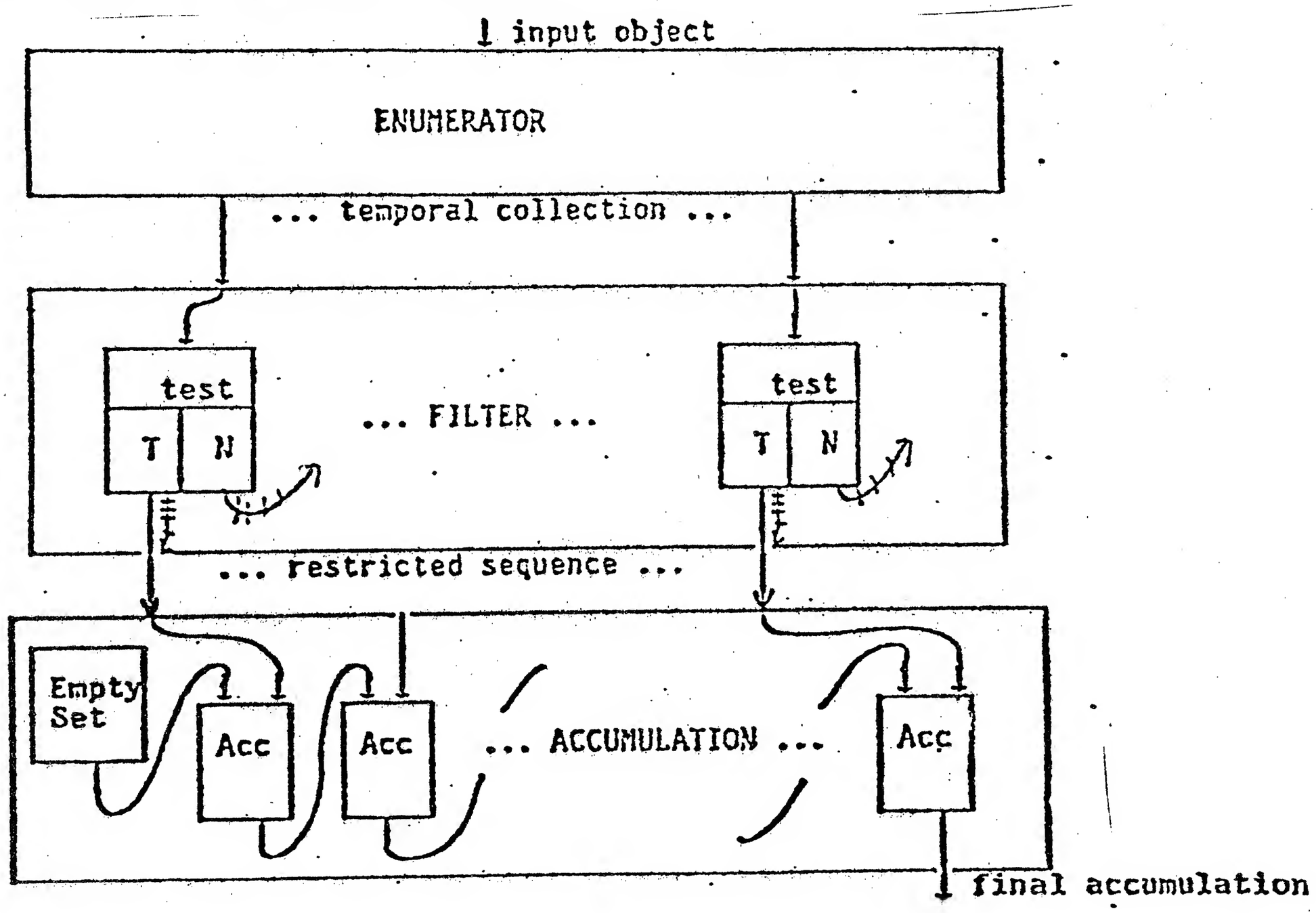

Figure 5: The plan diagram for the temporal model of Filtered Accumulation.

5. Queue and Process (Figure 6) This is characterized by it use of the queue. The purpose of this plan is to create abstract data flows between one occurrence of the body of the loop and later occurrences of the body. Recognition of this plan-type immediately suggests a temporal abstraction in which the abstract data flows are made manifest.

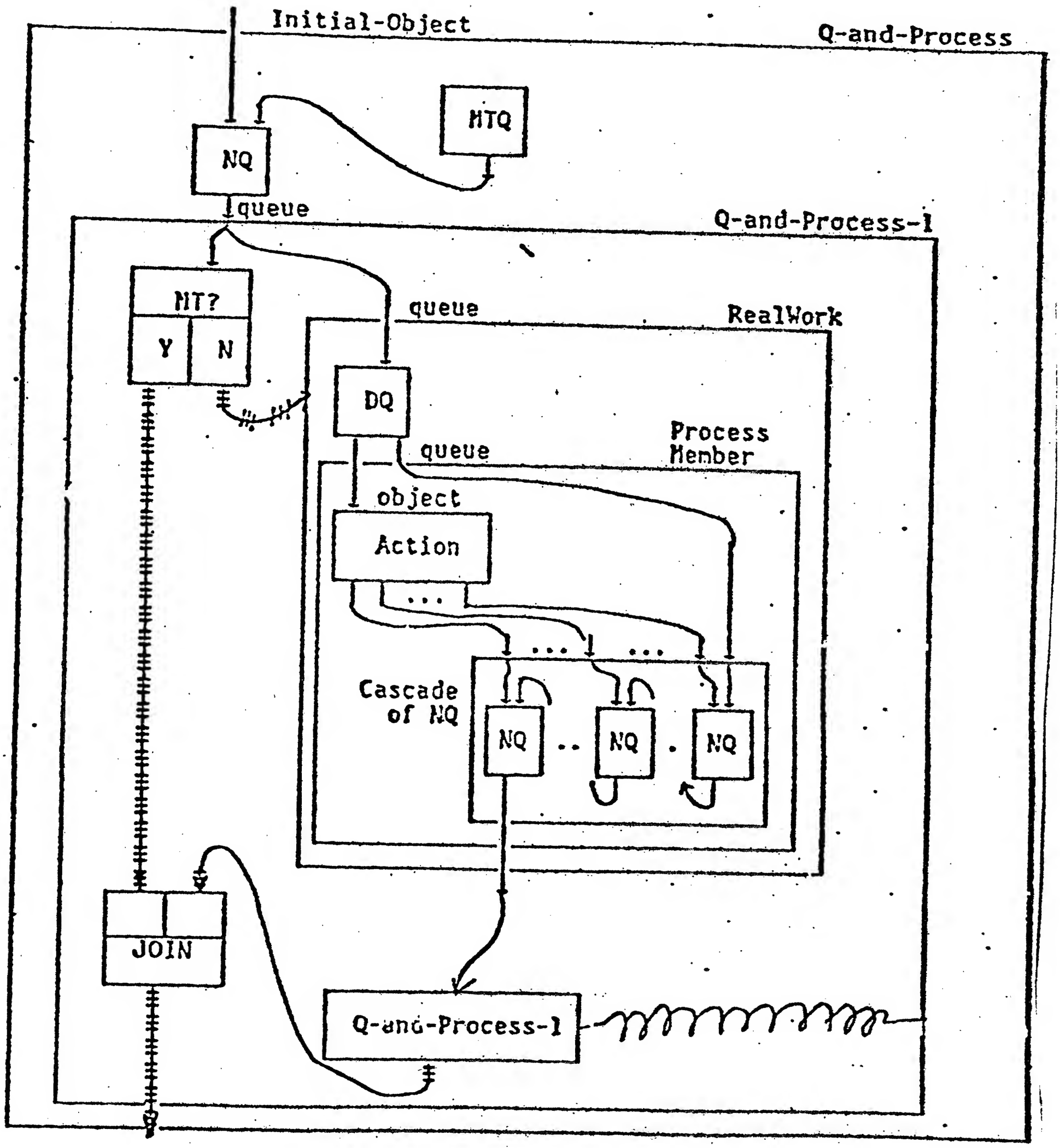

Figure 6: The plan diagram for the Queue and Process plan.

6. Hashing Plan (see figure 7) This is a prototype for the typical operations performed on a hash table. The plan is shown at a level of abstraction high enough so that it is valid for any of the standard implementations of hash-tables. It is characterized by a numerical calculation producing an index for an array reference. Frequently the last stage of the hash calculation is the computation of the absolute value of the remainder of some number when divided by the size of the array. The second segment of the hash plan fetches the appropriate bucket from the array. This bucket will be an implementation-dependent data structure which is used to represent a set. The last segment is the appropriate set operation specialized to the object-type of the bucket.

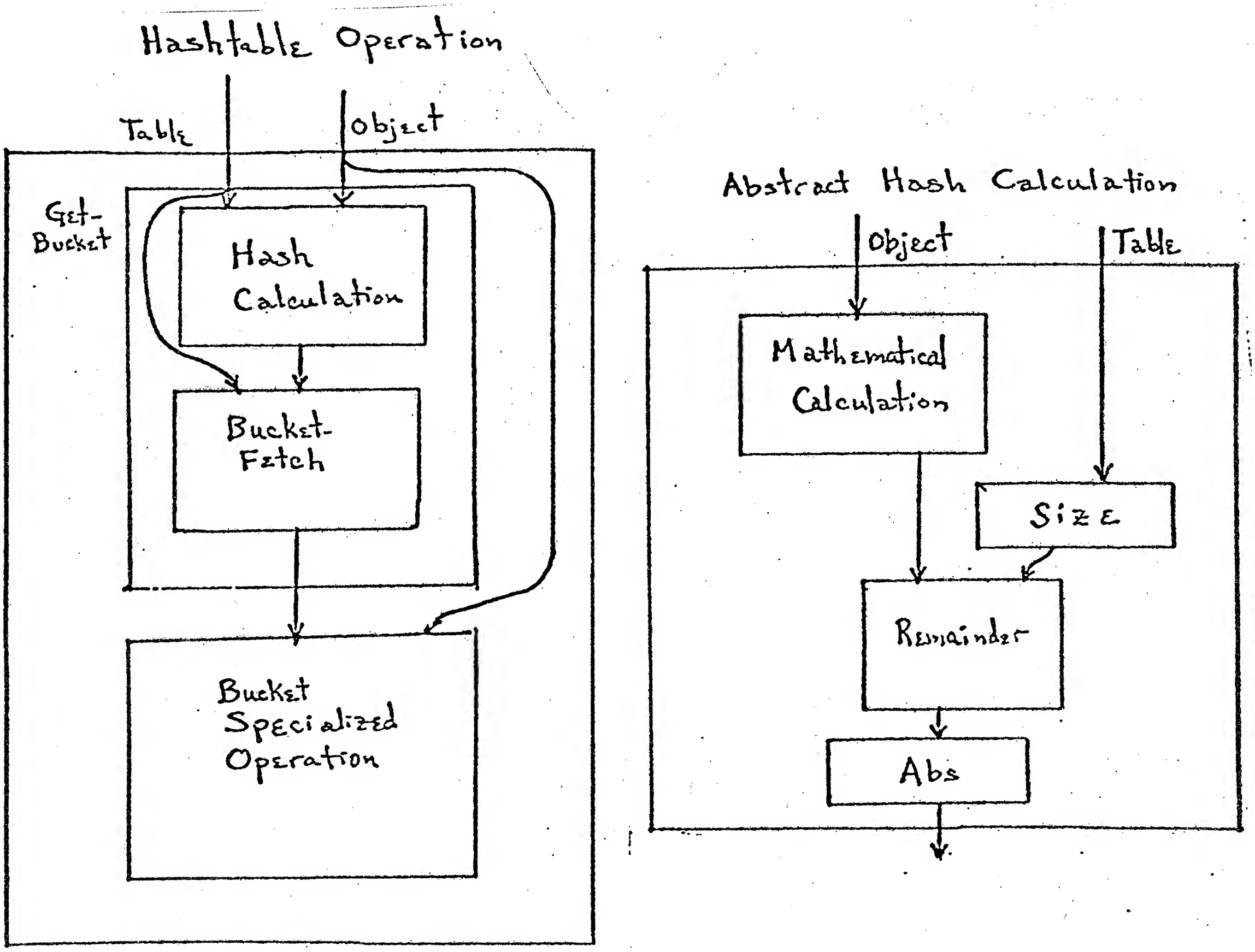

Figure 7: Hashing operations.

7. Linear Search (shown in Figure 8) This plan is most clearly shown in the temporal viewpoint. It consists of a cascade of test segments which test a temporal collection of values. One case of each test terminates execution, the other case enables the next test segment and returns control to the generator.

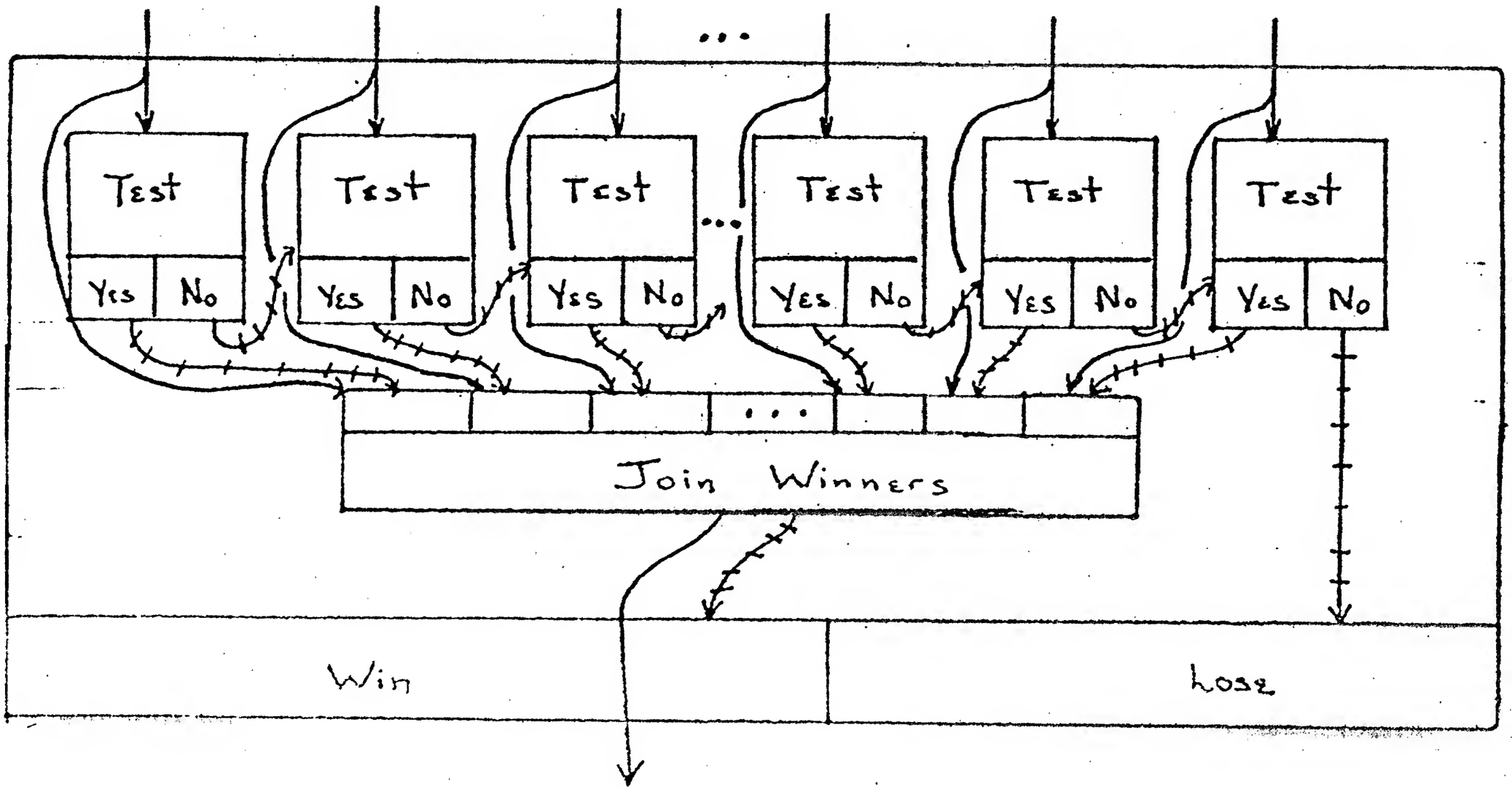

Figure 8: Plan diagram for Linear Search.

8. Trailing Pointer Enumeration - Splicing (Figure 9) Splicing plans are used to insert or delete elements from list structures. The generator part is called a Trailing Pointer Enumeration; it produces pairs of pointers to list cells (called the Current and Previous pointer) such that the current pointer points to the Cdr of the cell pointed to by the previous pointer. The splicing part of the plan is characterized by a Searching Plan and a side-effect which changes the CDR of the cell which caused the exit.

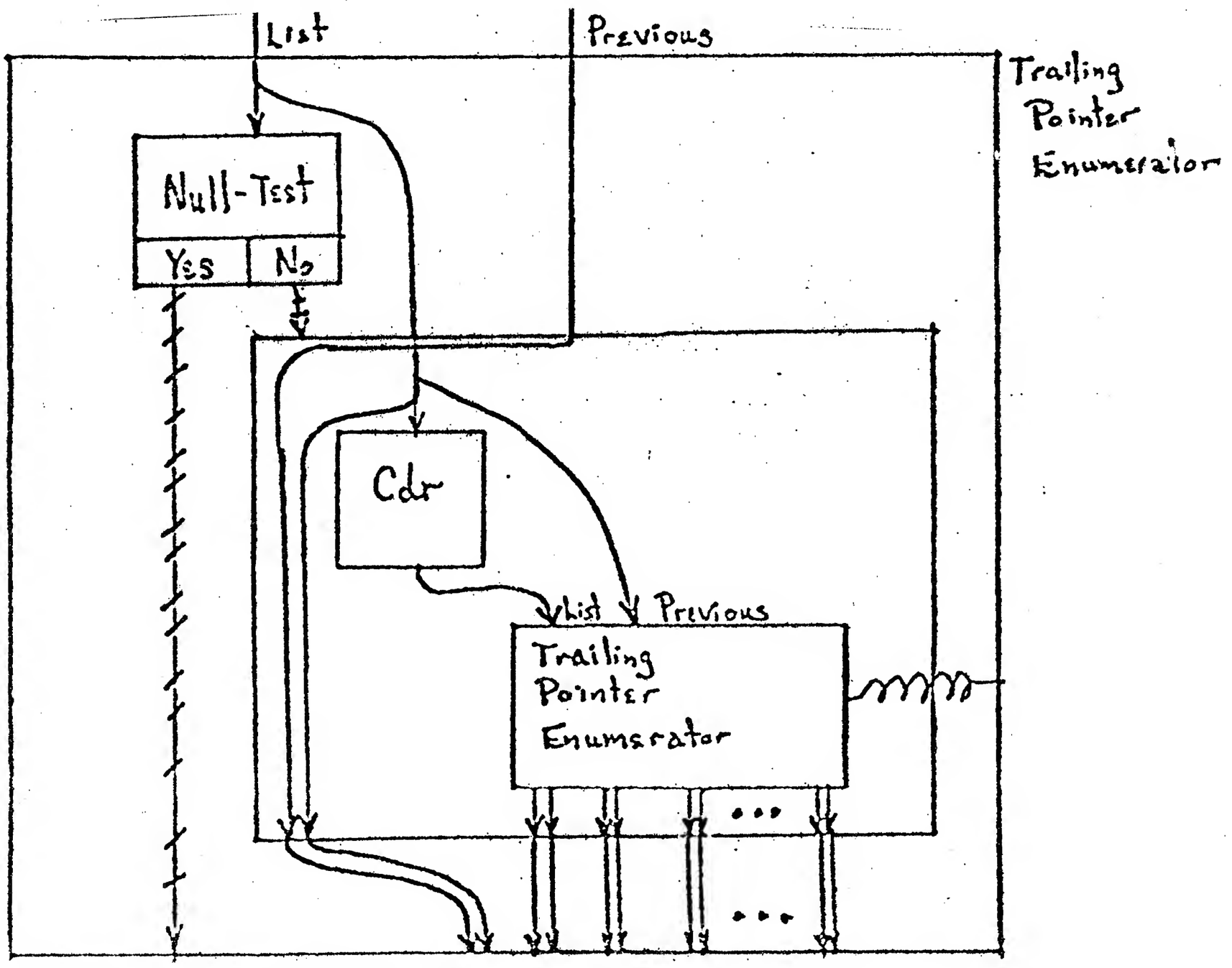

Figure 9: Plan diagram for Trailing Pointer Enumeration.

9. Enumerate and Side-Effect Each (Figure 10) This is a simple plan-type in which each element enumerated is subjected to the same side-effect.

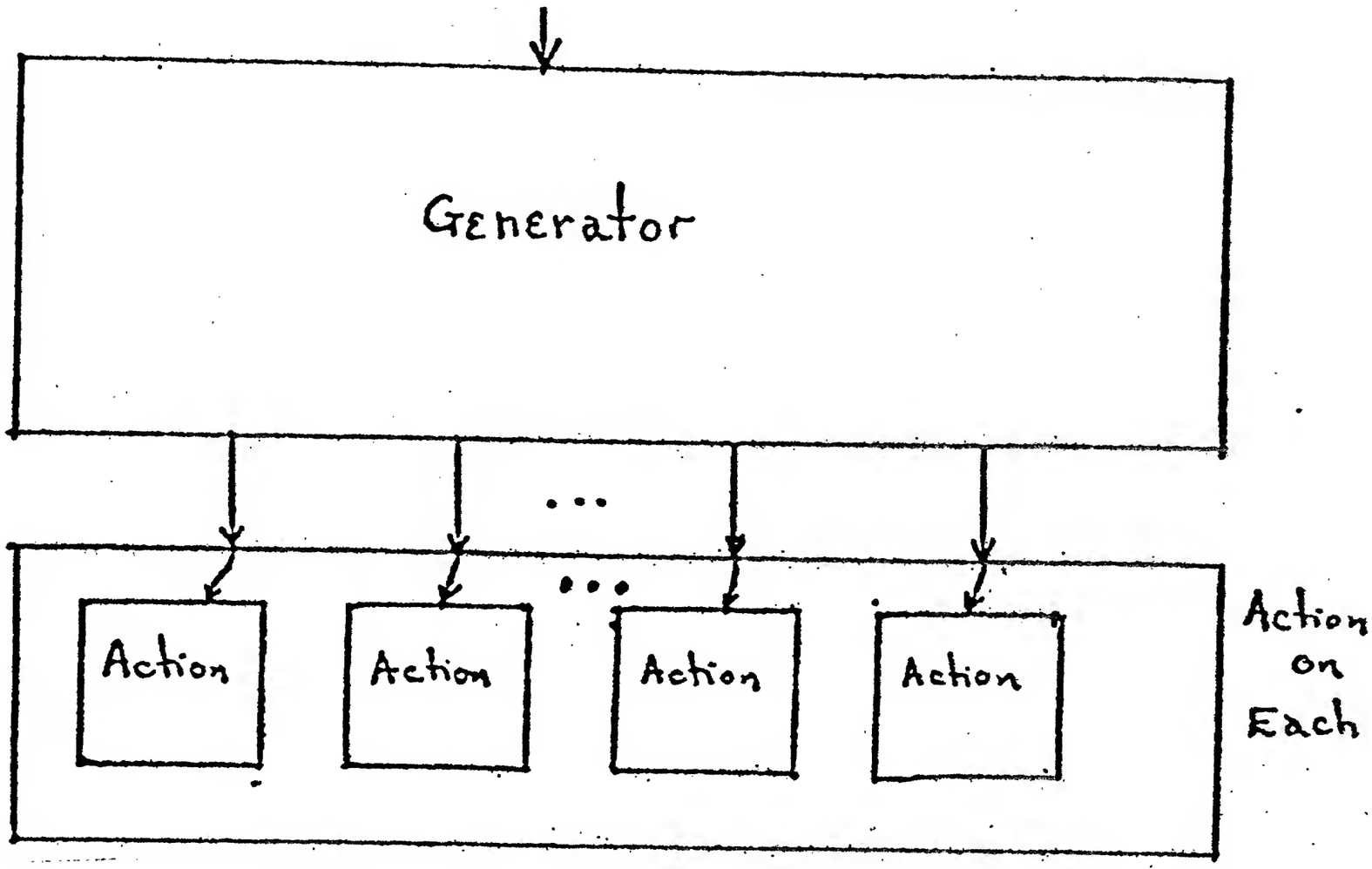

Figure 10: Plan diagram for Enumerate and Side-Effect Each.

10. Set Insertion (Figure 11) This is often used for sets which are represented explicitly in a data structure. The plan first checks to see if the object to be inserted is already present; the actual insertion into the data structure is only performed if the lookup does not find the object.

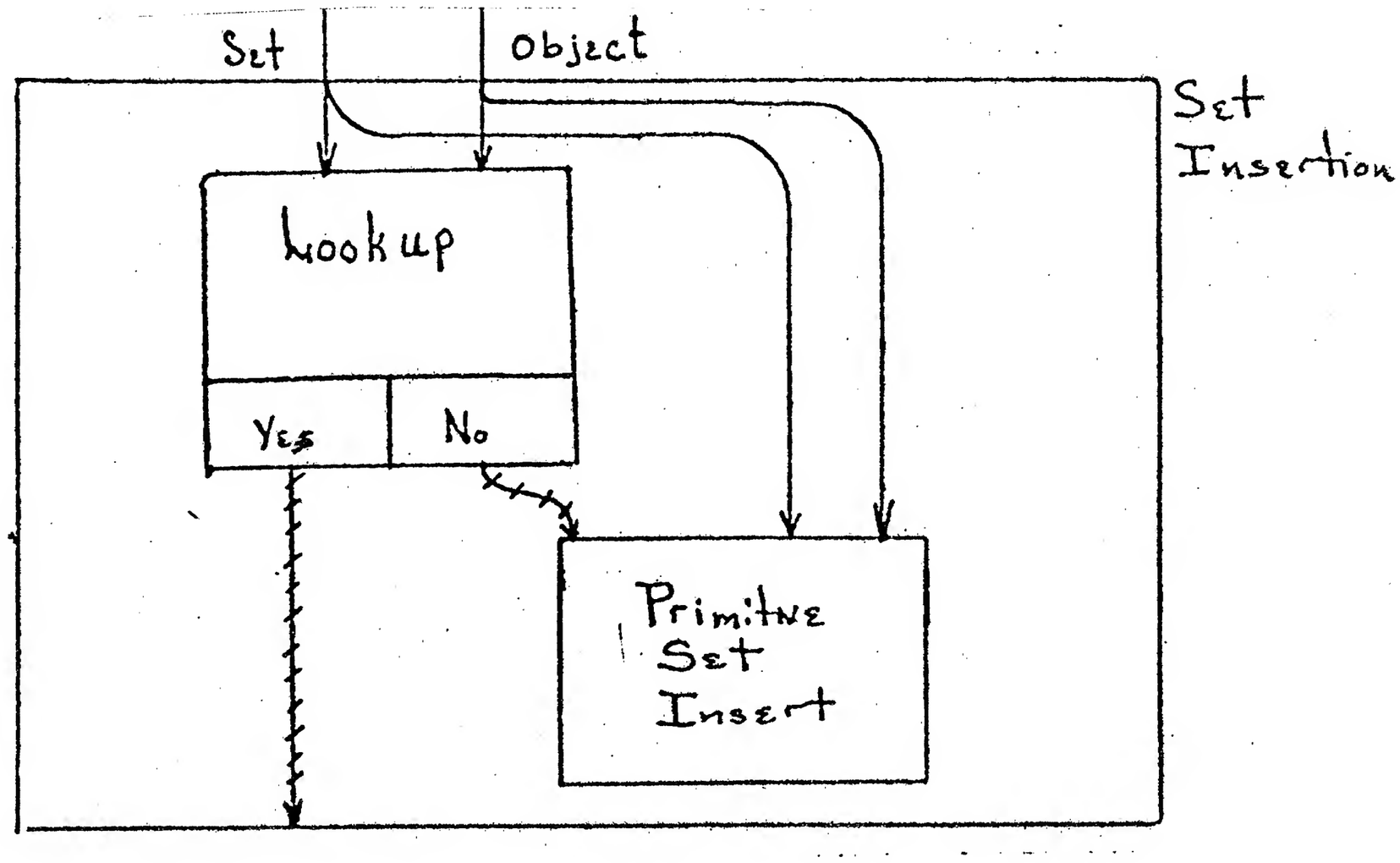

Figure 11: Plan for Set Insertion.

The plans shown so far constitute a set of techniques out of which many programs can be built. They are coordinated into higher level plans which depend significantly on the particular application domain. Such higher level organizing plans often exhibit an alternating layered structure. One layer represents an abstract data structure; the next layer operates on this structure so as to implement a data abstraction. The plan library is structured to reflect this pattern. It includes a vocabulary consisting of abstract data types annotated by the plans which are likely to operate on these types. Each type is also annotated by information indicating the typical implementation plans.

High level plans impose global structure on a program. For example, a data base can be regarded as a set which accumulates assertions. Thus, there is a point of view which presents the system as an accumulation plan with the data base acting as the accumulator. However it is unlikely that the system will have a single accumulation loop in which assertions are added to the data base. Typically, there will be many places in the program where assertions are added to the data base; the accumulation plan is distributed throughout the system and is part of the system's global structure. Temporal analysis can be used to identify such *global plans* and to model them as single coherent units.

The Monologue

This scenario is presented as a monologue reflecting the apprentice's reasoning processes as it reads the code for an example system of programs. Three fonts are used to distinguish the code, the apprentice's "train of thought" and our discussion of the aprentice's thinking.

THE CODE IS WRITTERN IN A SMALL CAPITALIZED FONT.

The apprentice "thinks" in the normal font used in the rest of the text.

[*The authors comment on the apprentice's thoughts in italics inside brackets*].

[*The apprentice begins to read the input file.*]

```
;;; A SIMPLE DEDUCTION SYSTEM
;;; IMPLEMENTED USING A PATTERN-DIRECTED DATA BASE AND ANTECEDENT RULES.
```

I know of only three ways of implementing pattern-directed data bases: hashing schemes, discrimination nets and property lists. Therefore I expect that one of these is being used. I will wait for triggering information to suggest a selection among these.

The comment says this is a deduction system, thus there must be some scheme for sequencing through the inferences to be made. I will expect to see an "agenda" mechanism for buffering actions to be taken. This may take the form of an explicit data structure (such as a queue for a system based on breadth-first search or an explicit stack for a system based on depth-first search) or the implicit stack of the implementation language if the system is recursively controlled.

```
(SETQ *DB*   (*ARRAY NIL T TABLESIZE)) ;this array holds the facts in the fact data base.
(SETQ *D-DB* (*ARRAY NIL T TABLESIZE)) ;this array holds the demons in the demon data base.
```

The particular code I see is the initialization of two arrays. The comments say these arrays are part of the data base. The only way I know to use arrays in AI data base systems is for hashing data objects. Thus I will assume that the system uses these arrays as hash tables. [*Notice that the apprentice used the array declarations as a "trigger fact" to jump to a conclusion.*]

```
;;; Functions for entering new facts and demons into the data bases
(DEFUN ENTER-FACT (FACT)
1       (COND ((NULL (LOOKUP FACT *DB*)) ;if not already in the db
2              (INSERT FACT)             ;put it in
3              (MAPC '(LAMBDA (DEMON)    ;and enqueue all the matching demons
4                             (ENQUEUE (LIST DEMON FACT) *TQ*))
5                     (DEMON-LOOKUP FACT *D-DB*)))))
```

[*When faced with a piece of code, the apprentice first breaks it up using PBM analysis and temporal abstraction. Then it begins to reason about the parts it has found.*]

The function name "LOOKUP" (used on line 1) is associated with data base plans. It indicates a module which can test whether a given fact is present in the data base. I assume that this is a Data Base Lookup. Lookup modules take at least two inputs, the fact and the data base. [*The apprentice talks about "inputs" which are part of the terminology of the surface plan representation. In the actual code inputs may be implemented as explicitly passed arguments (as is done here) or they may be implicitly passed through free variables or parts of global data structures referred to by free variables.*]

The function name "DEMON-LOOKUP" (line 5) also suggests a lookup routine, but this module takes a different data base as an argument. It would seem that LOOKUP determines whether a particular fact is in the data base *DB* and that DEMON-LOOKUP checks whether a particular demon is in the data base *D-DB*. However, the way in which these function are used indicates that they return lists. Lookup routines for associative data structures often return a set of items which match the input. Since an AI data base is an associative structure and since lists are often used to represent sets this is likely to be the case here.

I have already decided that there will probably be some Agenda which is used to buffer waiting actions. The comment on line 3 mentions an ENQUEUE and the function invoked on line 4 is named ENQUEUE; this suggests strongly that the agenda is being implemented by a queue. This means that there must be a Queue-and-Process plan to drive the agenda.

The fragment on lines 1 and 2 is a Set Insertion Plan which is often used in data base Insert routines. Therefore, I conclude that this is a data base INSERT fragment and also that one fragment of this plays the role of an ACTION-ENQUEUE part of a Queue-and-Process plan.

```
(DEFUN ENTER-DEMON (DEMON DB D-DB)
1       ;; if the demon is not already in the d-db
2       (COND ((NOT (MEMBER DEMON (LOOKUP (CAR DEMON) *D-DB*)))
3              (DEMON-INSERT DEMON)      ;put it in
4       ;; and apply the demon to all of the facts its pattern matches.
5              (MAPC '(LAMBDA (FACT) (ENQUEUE (LIST DEMON FACT) *TQ*))
6                    (LOOKUP (CAR DEMON) *DB*)))))
```

Again the ENQUEUE suggests the use of a Queue-and-Process plan. Obviously the queue contains pairs of demons and facts. Lines 2-3 are probably a Set Insert, with the fragment on line 2 containing (NOT (MEMBER DEMON (LOOKUP ...))) acting as the Test Part of the Set Insert plan. However, I don't yet understand exactly how it implements this behavior and I will not understand it until I understand exactly what LOOKUP does.

One thing I can say based on this fragment is that demons are indexed by their CARs.

Therefore CAR must select the PATTERN-PART of the data structure representing a demon. This is confirmed by line 6 where matching facts are looked up using the CAR of the demon as the FETCH-PATTERN.

[*Here we see an instance of the apprentice inferring the correct organizing structure, even though it cannot yet understand all of the fragments of the code in detail. We also see the apprentice attempting to discover the layout of the data structure which represents demons. As with the organizing structure, it can only infer part of the information needed.*]

```
;;; Basic insertion and retrieval functions for the data bases.
(DEFUN INSERT (FACT)
1       ;; insert the fact in data base.
2       ;; note: does not check if fact already present.
3       (MAPC '(LAMBDA (INDEX) (BUCKET-INSERT FACT INDEX *DB*))
4             (INDEX FACT *DB*)))
```

PBM analysis of the surface flow implemented by the MAPC on lines 3-4 indicates this is an Enumerate and Side-Effect Each plan whose enumerator is a List-Traversal.

The action performed on each member of the list is a BUCKET-INSERT. This suggests two things. First the use of the term BUCKET in this function name suggests that I was correct in assuming that the data base *DB* is a hashed system. Second, the use of the term INSERT suggests a Set-Insertion operation. Since the buckets of a hash table are constrained to be a data structure representing a set this is a consistent interpretation. I can infer from these assumptions, therefore, that INSERT makes FACT be a member of every bucket returned by the call to the function INDEX on line 4.

The output of the call to INDEX is fed into the MAPC, therefore I can tell that INDEX should return a list of elements which will be input to BUCKET-INSERT. The interface between INDEX and BUCKET-INSERT must be compatible, so if I find out that INDEX returns a list of numbers, I should expect BUCKET-INSERT to input a table-index (and vice-versa). Similarly, if INDEX returns a list of buckets, then BUCKET-INSERT must input a Bucket (and vice versa). There are various types of hashing systems, but in the one most commonly used in LISP, the buckets are implemented as lists. I will assume that this is the implementation used here, unless I find evidence to the contrary.

```
(DEFUN DEMON-INSERT (DEMON)
1       ;; insert demon in the demon data base.
2       ;; note: does not check if already present.
3       (MAPC '(LAMBDA (INDEX) (BUCKET-INSERT DEMON INDEX *D-DB*))
4             (INDEX (CAR DEMON) *D-DB*)))
```

This routine is virtually identical to the one above. The only difference is that the demon is only indexed by its CAR and that a different data base is used.

```
(DEFUN LOOKUP (PATTERN DB)
1       ;; return list of facts in data base which match pattern.
2       (PROG (INDICES  ;list of indices.
3              BKTS     ;list of buckets to be intersected.
4              HITS     ;list of facts in the intersection.
5              MATCHES) ;to accumulate facts that match.
6             (SETQ INDICES (INDEX PATTERN DB))
7             (SETQ BKTS (MAPCAR '(LAMBDA (INDEX) (ARRAYCALL T DB INDEX)) INDICES))
8             (SETQ BKTS (SORTCAR BKTS '<))
9             ;; intersect buckets to get candidates for match.
10            (SETQ HITS
11                  (DO ((L (CDR BKTS) (CDR L))
12                       (INT        ;to accumulate intersection.
13                        (CDAR BKTS) ;initialize to contents of first bucket.
14                        (FAST-INTERSECT INT (CDAR L))))
15                      ((OR (NULL L) (NULL INT))
16                       ;; done, return intersection.
17                       INT)))
18            ;; select only facts that match pattern.
19            (MAPC '(LAMBDA (FACT)
20                           (AND (MATCH FACT PATTERN)
21                                (SETQ MATCHES (CONS FACT MATCHES))))
22                  HITS)
23            (RETURN MATCHES)))
```

This module is quite a bit larger than the earlier ones. However, I can break this up into smaller units easily. The PBM analysis suggests that lines 10-17 are an Accumulation-Loop. Lines 6-8 form a unit which participates in both intializations for the loop. Its Accumulation Initialization is accomplished by taking the CDAR of the result of lines 6-8; the Enumeration Initialization is accomplished by taking the CDR of the result. Similarly PBM analysis indicates that lines 19-22 are a Filtered Accumulation Loop. Surface flow analysis indicates that the Enumeration Initialization of this loop is the loop on line 10-17. The Accumulation Initialization of the loop on lines 19-22 is on line 5. Finally, since line 7 is a MAPCAR, I can tell that it is an Accumulation Loop whose Enumeration Initialization is on line 6. The Accumulation Operator of the loop consists of the composition of the ARRAYCALL and a CONS which is implicit in the MAPCAR.

I already know that INDEX returns a list of either buckets or indices of buckets. The structure of this Filtered Accumulation Loop implies that each element of this list is input to the ARRAYCALL. Therefore, INDEX must return a list of hash table indices and it follows that BUCKET-INSERT expects its input to be a table index. Given the fact that this is a hashed data base, it would seem likely that the ARRAYCALL on line 7 is serving the role of a Bucket Fetch operation. For this to be a reasonable assumption it must be the case that the variable DB on line 7 is used to hold an array which is implementing a hashed data base. But flow analysis indicates

that this is an input variable of the module. So far, I have seen calls to LOOKUP in ENTER-FACT and ENTER-DEMON and flow analysis indicates that in each of these cases the object which flows to the DB input of LOOKUP is either *DB* or *D-DB*, both of which are arrays implementing Hashed Data Bases. Therefore, it follows that lines 6-7 produce a list of all the buckets whose indices are returned by (INDEX PATTERN DB).

The typical use of a Bucket Fetch operation within a Hashing Scheme involves calling it with a numerical operation which is the Hash of some object. I assume therefore that INDEX computes a list of Hashes. The fact that there are several Hashes is not typical, so I will have to examine INDEX to see exactly what Hashes are produced.

I have already decided that lines 10-17 constitute an unfiltered accumulation loop. The Accumulation Operation of this loop is the FAST-INTERSECT called on line 14. Since buckets are a particular representation of sets it is a reasonable assumption that FAST-INTERSECT computes the intersection of two sets. The initial accumulation is the first bucket in the list of buckets and the loop enumerates the rest of the buckets. Therefore, lines 10-17 computes the intersection of the collection of buckets computed on lines 6-8.

There are various facts which I cannot yet explain. FAST-INTERSECT is given the CDR of each bucket as its input, rather than the whole bucket. I will have to examine the structure of buckets to explain this. Also I do not yet understand the role of the sort on line 8. However, since it sorts using the CAR of the bucket as the Sort Key and since line 14 operates on the CDR of the bucket, it seems a likely assumption that buckets have two distinct conceptual parts represented by their CAR and their CDR. If my interpretation of FAST-INTERSECT is correct then it must be the case that the CDR of the list represents a set of items.

Finally the MAPC on lines 19-22 is a Filtered Accumulation Loop whose Filter is MATCH and whose Accumulation Operation is CONS. Since the Enumeration Input of this loop is the output of the previous loop, I can conclude that this loop builds a list of all those facts in the Set Intersection which also MATCH the input PATTERN.

The function name MATCH suggests the use of Pattern-Matching. I know a few varieties of pattern matching typical of AI systems, but I have no evidence to suggest one of these. The variable name PATTERN also suggests Pattern Matching so I assume that MATCH is a pattern matcher.

[Here we see a very clear example of how our techniques contribute to the analysis. Although each of the loops are coded in a different manner, Surface Flow Analysis reveals the commonality. The PBM analysis is used to separate the program into sub-units which can be analyzed in isolation to a large degree. The PBM annotation also determines a categorization of

these loops as filtered and unfiltered accumulations. This annotation leads to the construction of a temporal model in which the sequence of enumerated values is made explicit.

We also see the propagation of information between layers of annotation. Surface Flow Analysis is used to determine whether there are constraints on the type of the variable DB. This information is used to construct a recognition mapping to a standard plan in the library. The apprentice also uses such information to make further assumptions about what structures are likely to be seen later in the code.]

```
(DEFUN DEMON-LOOKUP (FACT D-DB)
1       ;; return list of demons in the demon data base whose pattern matches fact
2       (PROG (INDICES BKTS HITS MATCHES)
3             (SETQ INDICES (INDEX FACT D-DB))
4             (SETQ BKTS (MAPCAR '(LAMBDA (INDEX) (ARRAYCALL T D-DB INDEX)) INDICES))
5             (SETQ BKTS (SORTCAR BKTS '<))
6             ;; union buckets to get candidates for match.
7            (SETQ HITS
8                  (DO ((L (CDR BKTS) (CDR L))
9                       (INT (CDAR BKTS) (FAST-UNION INT (CDAR L))))
10                     ((NULL L) UNION)))
11            (MAPC '(LAMBDA (DEMON)
12                           (AND (MATCH FACT (CAR DEMON)) (SETQ MATCHES (CONS DEMON MATCHES))))
13                  HITS)
14            (RETURN MATCHES)))
```

This routine has a very similar structure to LOOKUP, but there is a difference. The last routine accumulated the Intersection of the indexed buckets, this one computes the Union. Why? I cannot yet ascribe any purpose to either the Intersection or the Union.

```
;;; Basic utility routines used by the data base functions
(DEFUN INDEX (FACT DB)
1       ;; returns list of indices for fact.
2       (PROG (*INDICES)
3             (INDEX1 FACT 1 DB)
4              (RETURN *INDICES)))
```

```
(DEFUN INDEX1 (FACT POSITION DB)
1       ;; compute indices and return them by
2       ;; cons'ing onto global variable *INDICES.
3       (COND ((ATOM FACT)
4              ;; note that the symbol "*" is not indexed.
5              (OR (EQ FACT '*)
6                  ((LAMBDA (INDEX)
7                           (OR (MEMQ INDEX *INDICES)
8                               (SETQ *INDICES (CONS INDEX *INDICES))))
9                    (ABS (REMAINDER (+ (MAKNUM FACT) (LSH POSITION 18.))
10                                   (CADR (ARRAYDIMS DB)))))))
11             (T
12              (INDEX1 (CAR FACT) (LSH POSITION 1))
13              (INDEX1 (CDR FACT) (1+ (LSH POSITION 1))))))
```

Temporal Abstraction reveals that these two modules contain a fragment which is a Tree Traversal whose input is the variable FACT. The recursion on the CAR is on line 12, the recursion on the CDR is on line 13. In parallel with this, there is a numerical calculation (represented by the LSH forms). However, I do not understand the role played by the numerical calculations. The Enumeration Input of the Tree Traversal is the variable Fact in function INDEX. This structure is reasonable only if the input Fact is regarded as a binary tree. Flow Analysis shows that the object which flows to the FACT input of INDEX is either a data base Fact or the Pattern Part of a Demon. I conclude that facts and Demon Patterns are regarded as Binary Trees. Since these objects also flow to the MATCH function, it seems likely that MATCH will be a Binary Tree Structure Matcher.

From Temporal Analysis I can also tell that the Atom Test on line 3 serves both as the Termination of the recursion and as a Filter which allows only the atomic nodes through to the Body. Temporal Analysis of lines 5-10 determines that this fragment is a Filtered Accumulation. Line 5 Filters out those Terminal Nodes which are the symbol "*". Lines 7 & 8 form a Set Insertion plan; line 8 alone would form a Sequential Cons Accumulation. This means that lines 7 and 8 together implement the accumulation of a Set as opposed to a Multi-Set.

The calculation on lines 9-10 is some arithmetic calculation which I do not understand. However, I see two suggestive features: The first is that it calculates the array's dimension. The second is that it returns a result which is the absolute value of the remainder of dividing by the array dimension. I have already noted that INDEX very likely calculates a set of Hashes and these features match my description of the Hash Calculation plan. I will assume, therefore, that this is the Hash Calculation. The results of the numerical calculations on lines 12 and 13 flow into the Hash Calculation. I will assume that they are related to the Hashing.

I now know that INDEX Traverses the Tree Structure of the input and Sequentially Conses up a Set of the Hashes of those Terminal Nodes which are not the symbol "*". I don't yet know why the symbol "*" is distinguished.

Now that I understand INDEX, I can conclude that the INSERT routine Inserts a fact into each member of a set of buckets. The buckets are those Hashed to by the Terminal Nodes of the fact. However, if a terminal node is the symbol "*", then the Fact isn't inserted into that Bucket. Similarly INSERT-DEMON Inserts a Demon in each member of a set of buckets. In this case the buckets are those Hashed to by the Terminal Nodes of the Pattern Part of the Demon. I know that a Data Base Insert routine is intended to make its input be a member of the data base. It seems reasonable to assume that the definition of membership in this data base is that an item is a member of the data base if it is a member of each bucket which is hashed to by one of its

terminal nodes. However, terminal nodes which are the symbol "*" are exceptions. Similarly I conclude that the definition of membership in the demon data base is that a demon is a member of the data base if it is a member of every bucket hashed to by a terminal node of its Pattern part.

This now lets me expand my understanding of the LOOKUP routine. Given a pattern, it fetches a bucket corresponding to each terminal node. But the assumption is that any member of the table must be a member of each of these buckets as well. Therefore, it must be in the intersection calculated by the Filtered Accumulation Loops in the Lookup routines. Lookup then filters these facts in the intersection by using the function Match which I have assumed is a Pattern Matcher. Therefore LOOKUP returns those facts which match a given pattern.

DEMON-LOOKUP is very much the same except that it indexes a Fact which I assume can't be a member of the demon data base and uses the indices generated to retrieve Demons. This indicates that something more complex is going on. The use of the function MATCH indicates that DEMON-LOOKUP returns Demons whose patterns match the given Fact. Presumably, it returns all such Demons.

```
(DEFUN MATCH (FACT PATTERN)
1       ;; predicate which returns T only if fact matches pattern.
2       ;; note: no variables are used, only * which matches anything.
3       (COND ((OR (ATOM FACT) (ATOM PATTERN))
4              (OR (EQ PATTERN '*) (EQ PATTERN FACT)))
5             (T ; tree recursion.
6              (AND (MATCH (CAR FACT) (CAR PATTERN))
7                   (MATCH (CDR FACT) (CDR PATTERN))))))
```

My previous conclusions have led me to expect that this routine will include a Binary Tree Traversal and this structure is obviously present. The two recursive calls are combined by the AND on line 6; the temporal model shows this to be a Search Plan. Line 3 indicates that this Search Plan operates on the Temporal Collection consisting of pairs of nodes - one from each Binary Tree - where at least one of the nodes is a terminal.

[*By the time the apprentice gets here, it has extremely strong expectations of what structure should be found in the pattern matcher. Notice the degree to which it can analyze this function using plan recognition. We omit the code for the FAST-INTERSECT and FAST-UNION routines; from its analysis of these two routines, the apprentice deduces that the buckets must be sorted in increasing order.*]

```
(DEFUN BUCKET-INSERT (FACT INDEX DB)
1       ;; splice fact into indexed bucket in maknum order.
2       ;; note: does not check for duplicates.
3       (PROG (BKT)
4        ;; bucket fetch.
5        (SETQ BKT (ARRAYCALL T DB INDEX))
6        (COND (BKT
7               ;; search for place in bucket.
8               (DO ((B (CDR BKT) (CDR B))       ;B is ordered list.
9                    (PREV BKT B))               ;PREV is pointer for RPLACD.
10                  ((OR (NULL B) (> (MAKNUM (CAR B)) (MAKNUM FACT)))
11                   ;; do insertion.
12                   (RPLACD PREV (CONS FACT B))
13                   ;; update bucket count.
14                   (RPLACA BKT (1+ (CAR BKT))))))
15              (T ; previously unused bucket.
16               ;; build new bucket with one entry.
17               (STORE (ARRAYCALL T DB INDEX) (LIST 1 FACT))))))
```

The comment on line 2 uses the term Splice; this suggests a Splicing Plan. I have already determined that this function should be a Set Insertion into the Bucket data structure. I also know that the CDR of a Bucket is a List sorted in increasing order. I know that a Splicing Plan can be used to Insert an item in a list while maintaining its order. For this to work here, the splicing plan must search for an element larger than the one to be inserted; when this is found the new element should be Spliced-In in front of the element which stopped the search.

Surface Flow and PBM analysis have produced enough information for me to easily tell that lines 8 through 14 are a Splicing Plan. The Trailing Pointer Enumeration is effected by lines 8 and 9 of the Do Construct. The first of the two tests on line 10 is the Termination Test of the Enumeration; the second test is the Search test. The Splice Operation is on line 12. The comments on lines 8, 9 and 13 confirm this analysis.

I still don't know what role is played by the CAR of a Bucket. However the fact that the CAR of a Bucket is an object playing some other role allows the Trailing Pointer Enumeration to initialize its Trailing Pointer to the first cell of the Bucket as is done on line 9.

Line 14 is in the body of the Splicing Plan, but there is no role of the Splicing Plan which I can ascribe to it. The comment on line 13 uses the term COUNT, suggesting a Counting Plan; but there is no structure resembling a Counting Plan here. Is there a global counting structure? Line 14 changes the CAR of the Bucket, by adding 1 to it. This suggests that the Car of a Bucket must be a number. I conclude that a Bucket has two parts: A Count and a Membership List.

The hash tables were initialized to contain all NILs when they were created. It seems possible that Bucket-Insert can be called with the tables in that state. I have not yet seen any other initialization code. Therefore, I will extend my understanding of Buckets to allow NIL to be a valid Bucket. If I do find initialization code I will revise this conclusion.

Lines 6 and 15 test whether or not the Bucket is NIL. If the Bucket is not NIL, I will assume

that it has been properly initialized and is correctly structured. Line 15 is executed when the Bucket is NIL, so I assume that it is the initilization code. Line 15 stores a new object into the Table. This new object is a List of the number 1 and the fact to be inserted. To maintain consistency I want to show that this is a well formed Bucket. The CAR of this List is a number and the CDR is a list of facts which are members of the Bucket. Finally, the length of this list is 1. So everything is consistent.

The fuller description of the structure of a Bucket which I have gained here lets me explain one other thing which I didn't understand. The Lookup routines sort their list of Buckets using the CARs of the Buckets as sort keys. Now I can see that this orders them by their lengths. However, I don't know why this is done.

```
(DEFUN RUN ()
1       (PROG (*TQ*)   ;queue of demons and facts to execute
2             (SETQ *TQ* (MAKE-EMPTY-QUEUE))
3             (DO ((INPUT (READ) (READ)))  ;read in an input
4                 ((EQ INPUT 'STOP))       ;if 'STOP all done
5                 (EVAL INPUT)             ;evaluate it (things get queued up
6                 (DO ((DEMON-FACT-PAIR (DEQUEUE *TQ*) (DEQUEUE *TQ*))) ;for each one
7                     ((EMPTY-QUEUE *TQ*))
8                     (APPLY (CDAR DEMON-FACT-PAIR)  ;apply the demon to the fact
9                            (CDR DEMON-FACT-PAIR)))))))
```

PBM analysis separates out line 2 as the initilization of the Queue-and-Process plan which is on line 5 and 6-9. The Action Part of the Queue-and-Process plan applies the body of a demon to a fact. This depends on my analysis of the structure of queue entries which I made during the analysis of the two functions ENTER-FACT and ENTER-DEMON.

I can recognize the outer loop as a Driver Loop Plan; in such plans, a READ segment plays the two roles of Initialization and Bump. The Termination test is on line 4. The body of the Driver Loop includes the form on line 5 and the loop on lines 6-9.

The only code I have seen which ENQUEUES items on the queue are the two functions ENTER-FACT and ENTER-DEMON. However, there is no code which calls these two functions. Also I can't tell what actions will result from the APPLY on line 8 unless I know what demons will be in the queue. Again I could only tell this if I knew who called ENTER-DEMON.

Oh Well! I've encountered the end of the file and there are still things which I don't understand. I guess its time to start asking the programmer about these. [*From here the scenerio would continue with interaction between the user and the apprentice.*]

Discussion

Clearly there is a large computational cost associated with an analysis on this scale. However, there are many payoffs. The apprentice's understanding of the structure and purpose of this code is quite deep. It has sufficient understanding to assimilate new information easily and to interact with the user in an intelligent and natural way. If the programmer wanted to edit the code, the appropriate parts of the code could be identified semantically (for example, the bump step of the counting loop which updates the size field of a bucket). More significantly, the apprentice could tell the programmer what the ramifications of such a change might be (for example, the sort in the lookup routine might order the buckets differently).

We imagine the scenario shown above as a batch process which would be both preceded and followed by interactive sessions with the user. The user might first develop the code using the apprentice as an interactive coding aid. The plan language would serve the role of a high-level documentation facility. If the code were developed in this manner, the apprentice would have much stronger guidance than is indicated in this scenario. However, it is still likely that, as in this example, there will be things which the apprentice can't analyze. In that event the apprentice will analyze what it can and then query the user at its next opportunity.

A new generation of personal computers of considerable power is beginning to appear. It would not be unreasonable to give such a machine several hours to study the code above. It is quite likely that this would be an adequate time frame. We like to draw the analogy between the apprentice system and a junior colleague. Certainly, if one had a human assistant one would give him at least the morning to study the code in the example before giving him further assignments. The apprentice system would function similarly. The price of developing and maintaining software is the dominant cost of computation. As new generations of machines which provide increasing power for lower prices continue to appear the relative price of software will grow even more. Apprentice-like analysis systems will not solve all the problems in managing the complexity of the programming process, but they could make a qualitative reduction in the difficulty and expense of software maintenance.

References

Barstow, David [1977], "Automatic Construction of Algorithms and Data Structures", PhD. Thesis, Stanford University, September 1977.

de Kleer, J.; Doyle, J.; Steele, G.; and Sussman, G.J. [1977], "AMORD: Explicit Control of Reasoning", Proc. of the Symp. on AI and Programming Languages, August 1977.

Doyle, Jon [1978], "Truth Maintenance Systems for Problem Solving", MIT/AI/TR-419, January 1978.

Rich, C. [forthcomming], "A Library of Programming Plans with Applications to Automated Analysis, Synthesis and Verification of Programs", forthcoming PhD thesis, MIT Cambridge MA, expected 1979.

Rich C. and Shrobe H. [1976], "An Initial Report on a LISP Programmer's Apprentice", MIT/AI/TR-354, December 1976.

Rich, C. and Shrobe, H. [1978], "Initial Report on a LISP Programmer's Apprentice", IEEE Trans. on Soft. Eng., V4 #6, November 1978, pp. 456-467.

Shrobe, Howard E. [1978], "Reasoning and Logic for Complex Program Understanding", PhD thesis, MIT, August 1978.

Waters, Richard C. [1978], "A Method for Automatically Analyzing the Logical Structure of Programs", PhD thesis, MIT Cambridge MA, August 1978, (to appear as MIT/AI/TR-492).